Kennesaw State

Parser for EIL

Ethan Hunt

CS 4308 Concepts of Programing languages

Spring semester 2022

Sharon Perry

3/20/2022

Contents

[**Project Statement:** 1](#_Toc98687401)

[**EIL Language Updates: EIL v1.0** 2](#_Toc98687402)

[**LOOP Updates** 2](#_Toc98687403)

[**LOOP Construct structure** 2](#_Toc98687404)

[**Embedded LOOP Construct structure** 2](#_Toc98687405)

[**EIL If statements** 2](#_Toc98687406)

[**IF structure** 3](#_Toc98687407)

[**Report: Building a Parser for EIL in C++** 3](#_Toc98687408)

[**Phase One: Creating the parsing Algorithm** 4](#_Toc98687409)

[**Phase Two: Level Styled Parsing** 4](#_Toc98687410)

[**Output of Parser for various inputs** 5](#_Toc98687411)

# **Project Statement:**

The project is 100% complete will all parts working. In addition, basic interpreter functions are in the script as the way the interpreter is structured, I have decided to keep it in the deliverable, this however adds validity to the parser and its implementation. The Interpreter will also work without errors but has limited range of functionality for now as imbedded IF statements still need to be added. Note no new object files were made for the parser all new object files are made for the interpreter. Any code additions will be found in main.cpp, look for comments about parser functions.

# **EIL Language Updates: EIL v1.0**

EIL v1.0 has now been developed and contains more functionality than its beta and alpha versions. These additions included the ability to have different multi statement constructs interact with one another instead of the static application of the original constructs introduced in v0.5.

## **LOOP Updates**

EIL supports only one Loop type structure which as of EIL v0.5 has been updated in terms of syntax to the following structure seen in the LOOP Construct structure. The functions of the loop stay the same as a static copy of a variable’s value is used to determine how many times to loop the code inside. The syntax was designed to allow for simple intuitive abstract parsing techniques to implemented on the language. By switching to a universal statement closer of <;> in EIL v1.0 for the first round of statement parsing, allows us to only need to look for one possible delimiter. In addition to make less extensive stack calls on recursive loop structures only one embed loop is allowed. This is a new addition included in the release of EIL v1.0

### **LOOP Construct structure**

*<L> <ID>*

*<statement> <:>*

*….*

*<;>*

### **Embedded LOOP Construct structure**

*<L> <ID>*

*<L> <ID>*

*<statement> <)>*

*<:> <;>*

## **EIL If statements**

The IF conditional structure was added into EIL v1.0 to serve as the conditional structure in the language. It does not have a corresponding else statement instead an additional statement that has the opposite conditional will serve as the else statements. Impeded If statements have not been developed a will come out in a future version of EIL.

### **IF structure**

<IF> <ID> <Double\_Operator> <ID>

<statement> <$>

<;>

# **Report: Building a Parser for EIL in C++**

Similar to how I built the scanner I decided to build the parser without any additional resources regarding algorithms and methodologies for a parser. With these additional challenges I imposed on myself, I created my own method of parsing the program and modified EIL to meet those requirements focusing on EIL’s abilities to abstract statements and using that abstraction to my advantage. I was able to with this parsing technique very effectively create a simple interpreter in little time after the first time of statement parsing was finished. I have been slowly adding both new parser functions and interpreter functions such that if the statement can be parsed it can be interpreted as well.

## **Phase One: Creating the parsing Algorithm**

At first, I was a little intimidated as what I knew about parsers and parse trees seemed like it was going to be difficult to code as if I had to create node tress for each of the parts of the parser then it would become very resource heavy very fast. So, I instead took the approach to look at the parse tree as a sort of decision tree in which it could be implemented using If statements and finite state machines. The only issue with this method is how to store the information determined by the different conditionals. I ended on a queue in which I maintained that queue using to different stacks as this would allow me full functionality over queue process vs using stander queue structure.

## **Phase Two: Level Styled Parsing**

I stated the implementation of the parser by making that every statement of same level must end with the same delimiter token. A level is defined as grouping of statements contained within a larger parent statement. So, for all statements on level 1 (all statements that are not the child of some other statement) must end with the following <;> token equivalent. Take the following code for example:

Level Key: Color lexemes represent the start and end of a statement of a color level

Red: Level 0

Green: Level 1

Blue: LOOP Level

Purple: IF Level

Start M

a = 2 ;

c = 12 ;

L c

x = a \* x +1 :

b = 2 \* a :

IF x == b

x = 0 $

: ;

End M

As shown in the example above by looking at the first and last token in statement we can then take the middle and parse that section by a different token giving an easy and intuitive way to separate the different lines of code embedded within another statement. This can be implemented by simple popping of the front of a queue and evaluating the contents until a delimiter token is found. Then once that queue of tokens has been properly, pop the next queue of tokens out of the larger queue and repeat assessment. Once that queue level has been cleared move onto the next level until we need to change levels again. This goes on until we have parsed every single command above level 0.

# **Output of Parser for various inputs**

**The output of the following code EIL program**

**Start M**

**a = 0 ; b = 2 ; c = 3 ; x = 1 ;**

**L c**

**L b**

**a = a + 1 ) :**

**d = d + a + 1 : ;**

**P ;**

**End M**

Text

Description automatically generated

**The output of the following EIL program**

Note: This example has an intended errors in the code to show the parser handling the errors

**Start M**

**a = 0 ;**

**L 1**

**a x + 1 :**

**;**

**P ;**

**End M**

Text

Description automatically generated with medium confidence

**The output of the following code EIL program**

*Start M*

*a = 0 ; c = 3 ; b = 3 ;*

*L c*

*IF a == b*

*a = 0 $ :*

*a = a + 1 : ;*

*P ;*

*End M*

Text

Description automatically generated

# **Pictures / Lists of all Parser functions**

These are just main function screen shots so you can see what functions what are. The comment in the top left of each image tells function grouping. ex // Parser functions

Text

Description automatically generatedText

Description automatically generated

A picture containing text

Description automatically generated